RAMA UNIVERSITY

www.ramauniversity.ac.in

FACULTY OF ENGINEERING & TECHNOLOGY

Lecture -05 : Heap Short (Part-2)

Mr. Nilesh
Assistant Professor
Computer Science & Engineering

Outline

- Heap Short

Running Time of BUILD MAX HEAP

Alg: BUILD-MAX-HEAP(A)

1. n = length[A]
2. **for** i ← $\lfloor n/2 \rfloor$ **downto 1**
3. **do** MAX-HEAPIFY(A, i, n)

$$\left.\begin{array}{l} \\ O(\lg n) \end{array}\right\} O(n)$$

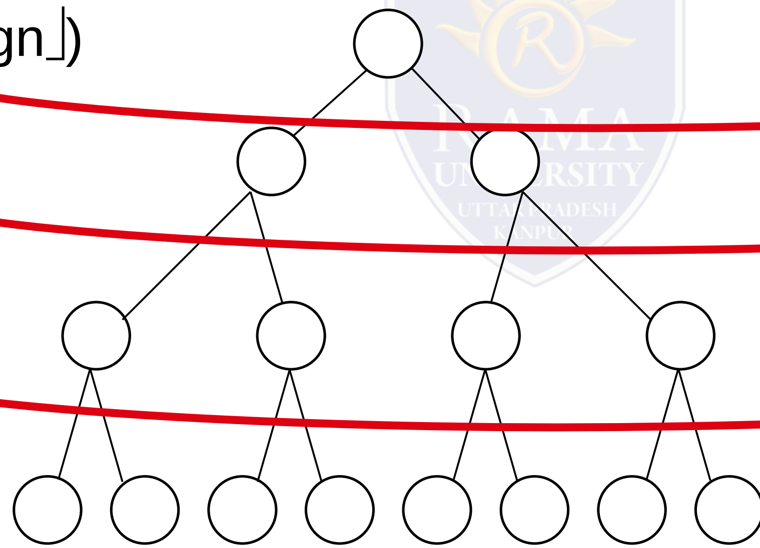
⇒ Running time: $O(n\lg n)$

- This is not an asymptotically tight upper bound

- HEAPIFY takes O(h) $\Rightarrow$ the cost of HEAPIFY on a node i is proportional to the height of the node i in the tree

$$\Rightarrow T(n) = \sum_{i=0}^{h} n_i h_i = \sum_{i=0}^{h} 2^i (h-i) = O(n)$$

Height	Level	No. of nodes
$h_0 = 3$ ($\lfloor \lg n \rfloor$)	i = 0	2^0
$h_1 = 2$	i = 1	2^1
$h_2 = 1$	i = 2	2^2
$h_3 = 0$	i = 3 ($\lfloor \lg n \rfloor$)	2^3



$h_i = h - i$ height of the heap rooted at level i

$n_i = 2^i$ number of nodes at level i

Running Time of BUILD MAX HEAP

$$T(n) = \sum_{i=0}^{h} n_i h_i$$ Cost of HEAPIFY at level i ∗ number of nodes at that level

$$= \sum_{i=0}^{h} 2^i (h - i)$$ Replace the values of n_i and h_i computed before

$$= \sum_{i=0}^{h} \frac{h-i}{2^{h-i}} 2^h$$ Multiply by 2^h both at the nominator and denominator and write 2^i as $\frac{1}{2^{-i}}$

$$= 2^h \sum_{k=0}^{h} \frac{k}{2^k}$$ Change variables: k = h - i

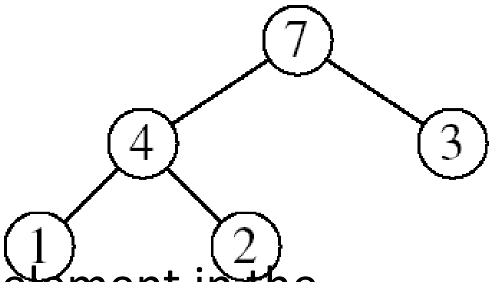
$$\le n \sum_{k=0}^{\infty} \frac{k}{2^k}$$ The sum above is smaller than the sum of all elements to ∞ and h = lgn

$$= O(n)$$ The sum above is smaller than 2

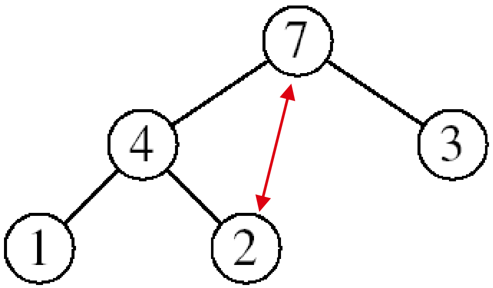
Running time of BUILD-MAX-HEAP: T(n) = O(n)

Heapsort

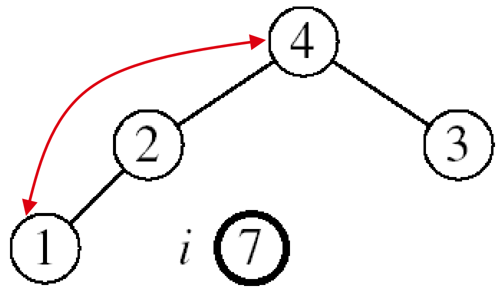
- Goal:
  - Sort an array using heap representations
- Idea:
  - Build a **max-heap** from the array
  - Swap the root (the maximum element) with the last element in the array
  - "Discard" this last node by decreasing the heap size
  - Call MAX-HEAPIFY on the new root
  - Repeat this process until only one node remains



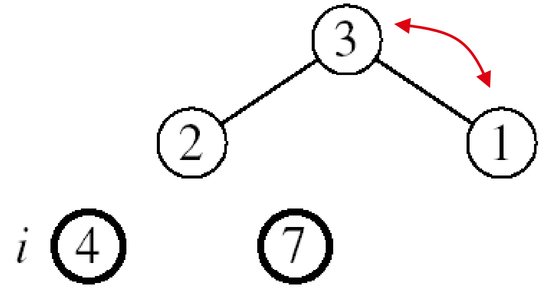
Example: A=[7, 4, 3, 1, 2]



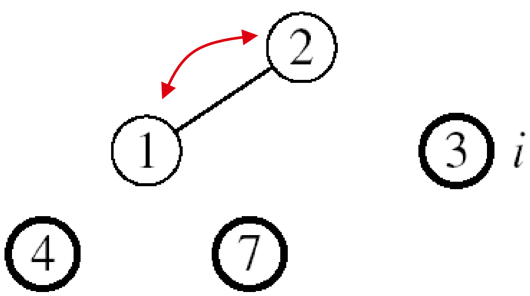
MAX-HEAPIFY(A, 1, 4)

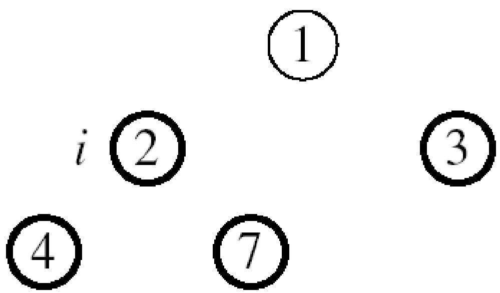


MAX-HEAPIFY(A, 1, 3)



MAX-HEAPIFY(A, 1, 2)





A					
	1	2	3	4	7

MAX-HEAPIFY(A, 1, 1)

Alg: HEAPSORT(*A*)

1. BUILD-MAX-HEAP(A)
2. **for** $i \leftarrow length[A]$ **downto** 2
3. **do** exchange $A[1] \leftrightarrow A[i]$
4. MAX-HEAPIFY(A, 1, i - 1)

$O(n)$

$O(\lg n)$

n-1 times

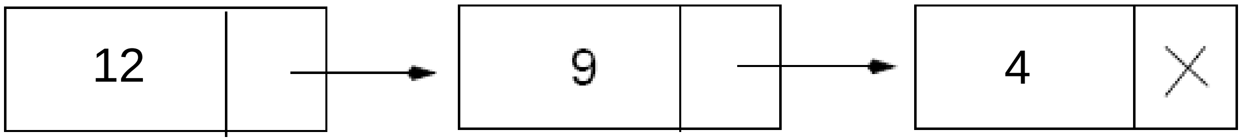
- Running time: $O(n\lg n)$ --- Can be shown to be $\Theta(n\lg n)$

Priority Queues

Properties

- Each element is associated with a value (priority)

- The key with the highest (or lowest) priority is extracted first



Operations on Priority Queues

- Max-priority queues support the following operations:
  - INSERT(S, x): inserts element x into set S
  - EXTRACT-MAX(S): removes and returns element of S with largest key
  - MAXIMUM(S): returns element of S with largest key
  - INCREASE-KEY(S, x, k): increases value of element x's key to k

    (Assume $k \geq x$'s current key value)

Goal:

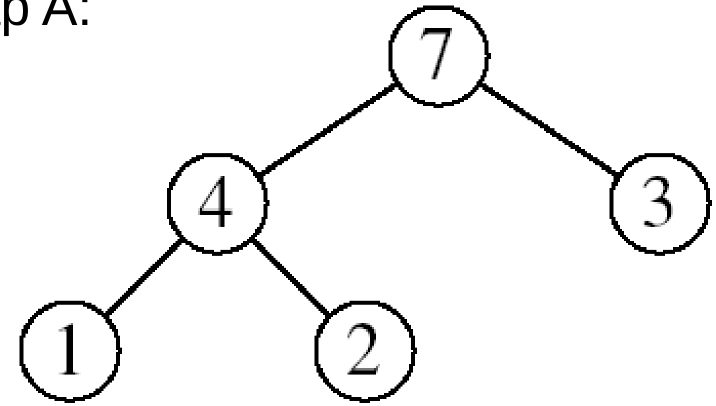
– Return the largest element of the heap

Alg: HEAP-MAXIMUM(A)

1. **return** $A[1]$

Running time: $O(1)$

Heap A:



Heap-Maximum(A) returns 7

HEAP-EXTRACT-MAX

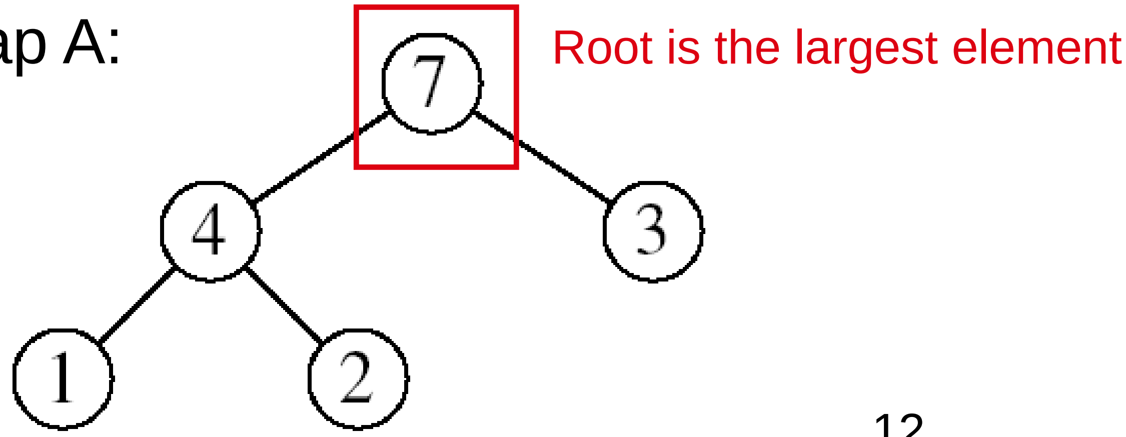
Goal:

- Extract the largest element of the heap (i.e., return the max value and also remove that element from the heap

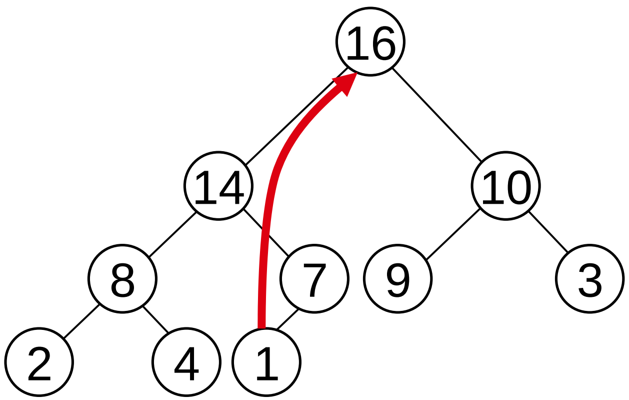
Idea:

- Exchange the root element with the last
- Decrease the size of the heap by 1 element
- Call MAX-HEAPIFY on the new root, on a heap of size n-1

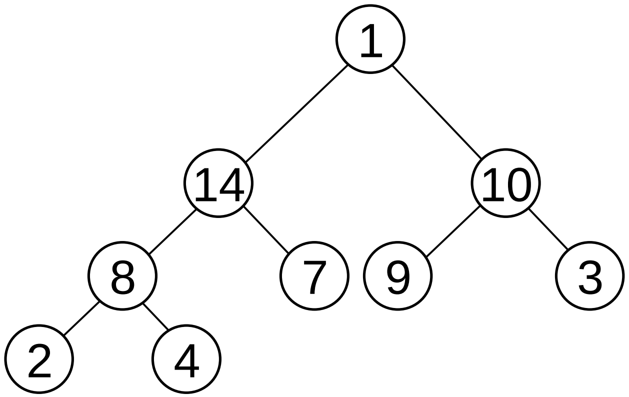
Heap A:



Example: HEAP-EXTRACT-MAX

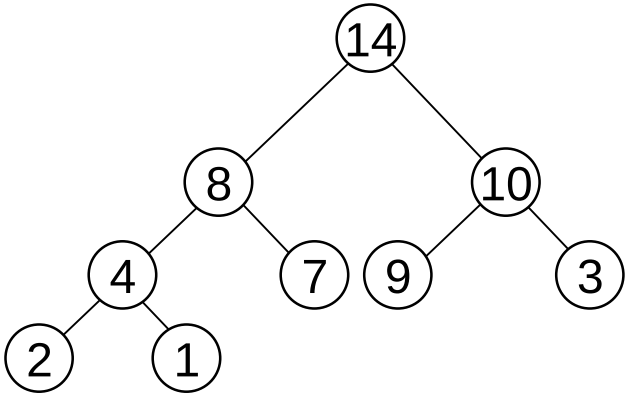


max = 16



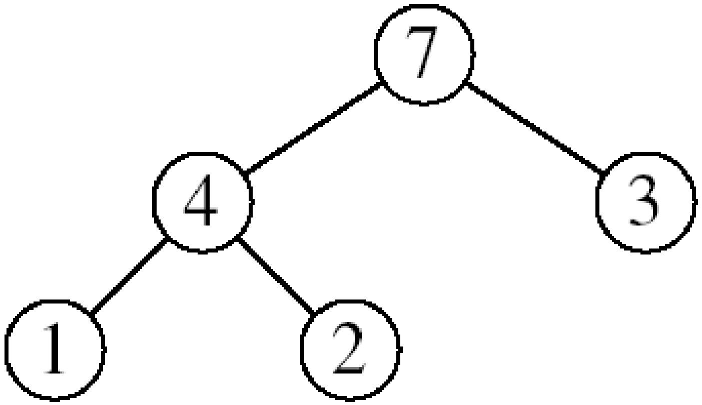
Heap size decreased with 1

Call MAX-HEAPIFY(A, 1, n-1)



Alg: HEAP-EXTRACT-MAX(A, n)

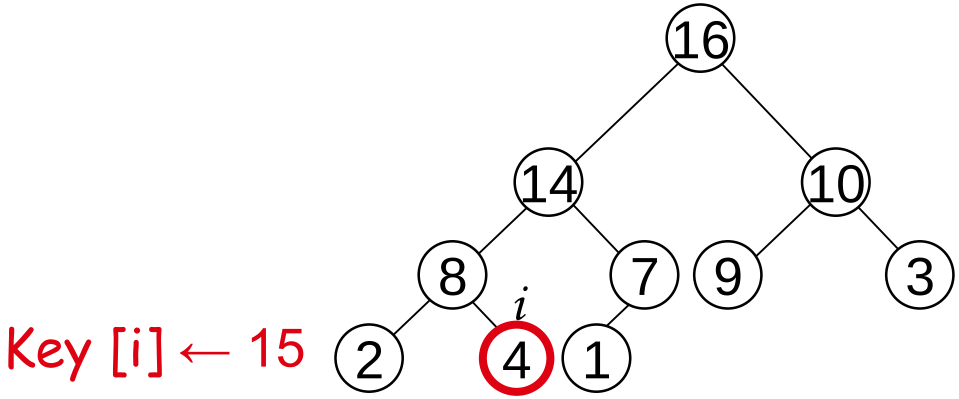
1. **if** $n < 1$
2. **then error** "heap underflow"
3. $max \leftarrow A[1]$
4. $A[1] \leftarrow A[n]$
5. MAX-HEAPIFY(A, 1, n-1) ▷ remakes heap
6. **return** max



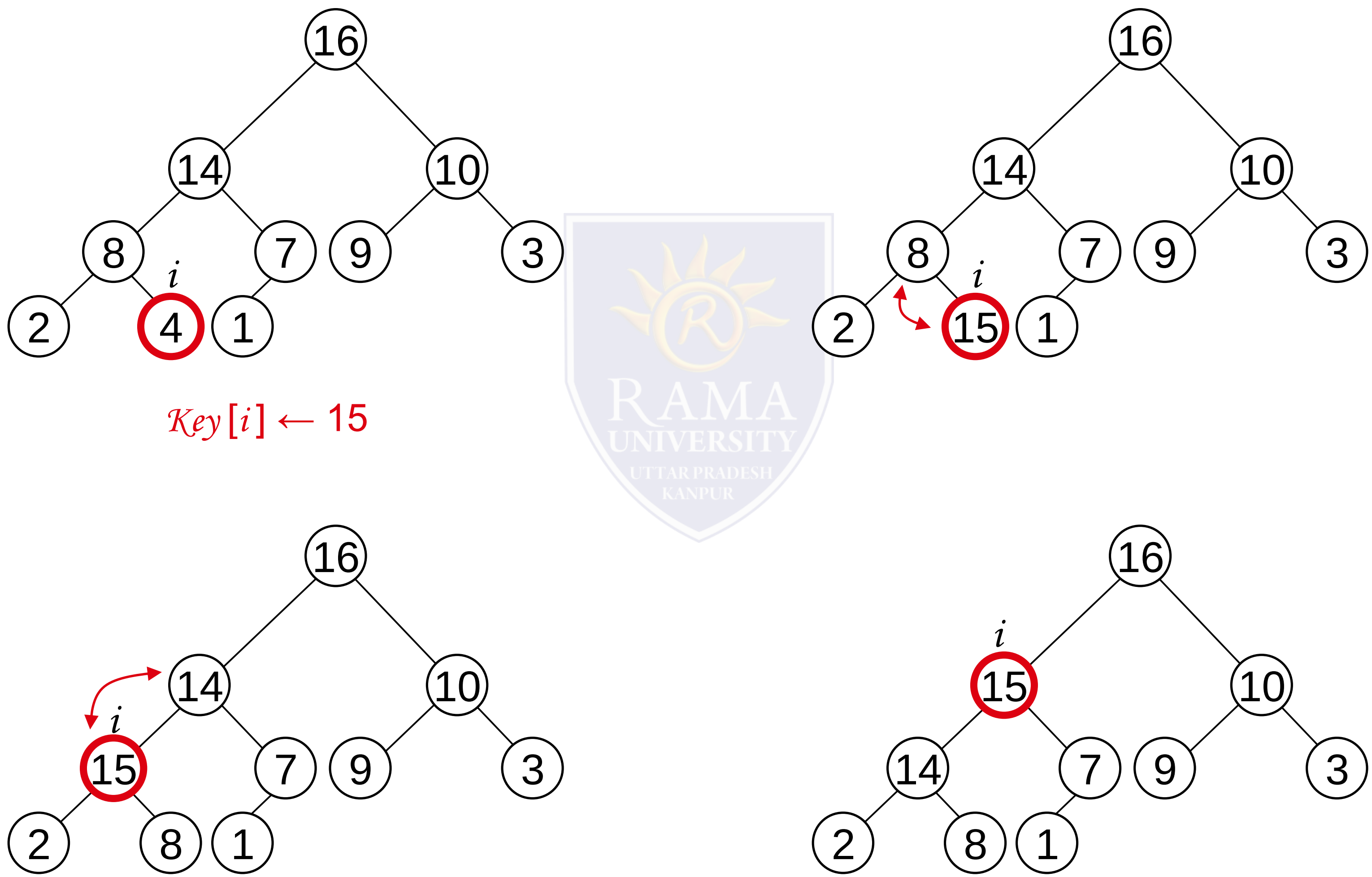
Running time: $O(\lg n)$

HEAP-INCREASE-KEY

- Goal:
  - Increases the key of an element i in the heap
- Idea:
  - Increment the key of $A[i]$ to its new value
  - If the max-heap property does not hold anymore: traverse a path toward the root to find the proper place for the newly increased key



Example: HEAP-INCREASE-KEY

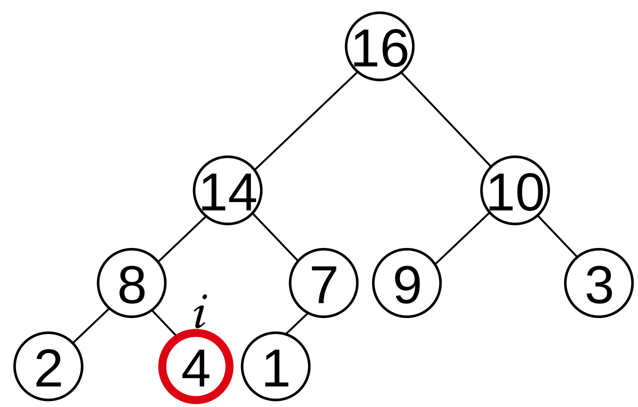


HEAP-INCREASE-KEY

Alg: HEAP-INCREASE-KEY(A, i, key)

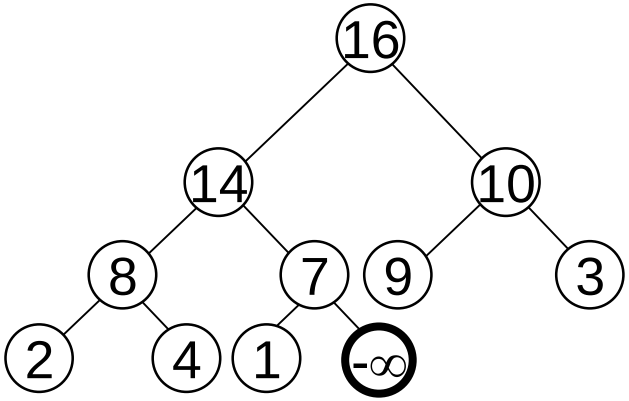
```
if key < A[i]
   then error "new key is smaller than current key"
A[i] ← key
while i > 1 and A[PARENT(i)] < A[i]
    do exchange A[i] ↔ A[PARENT(i)]
        i ← PARENT(i)
```

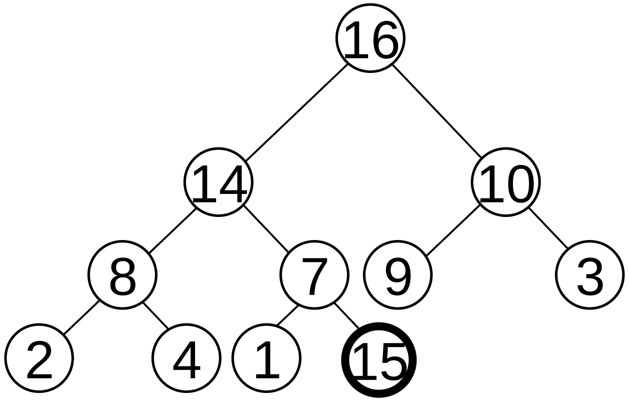
- Running time: $O(\lg n)$



Key [i] ← 15

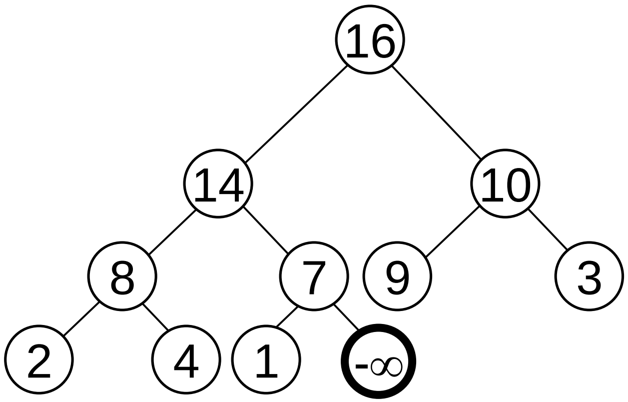
- Goal:
  - Inserts a new element into a max-heap
- Idea:
  - Expand the max-heap with a new element whose key is -∞
  - Calls HEAP-INCREASE-KEY to set the key of the new node to its correct value and maintain the max-heap property



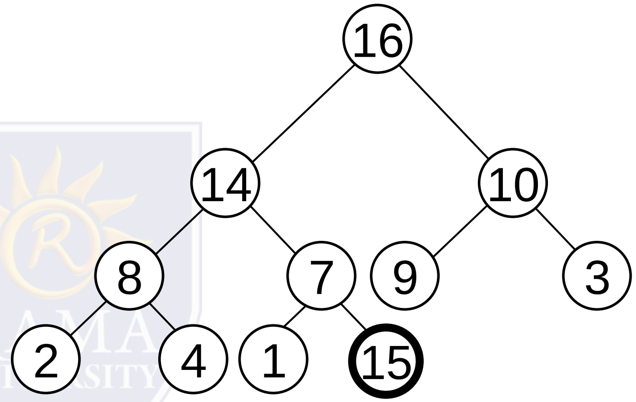


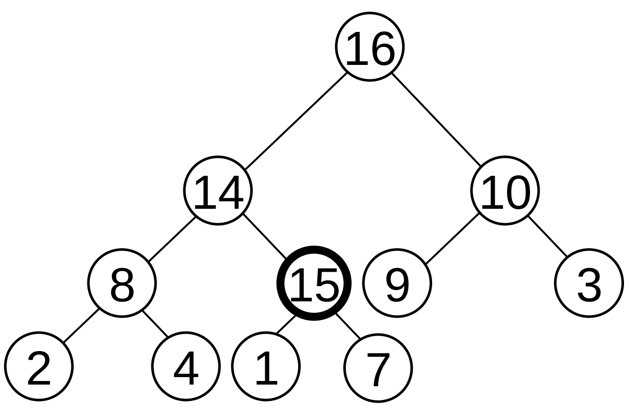
Example: MAX-HEAP-INSERT

Insert value 15:
- Start by inserting -∞

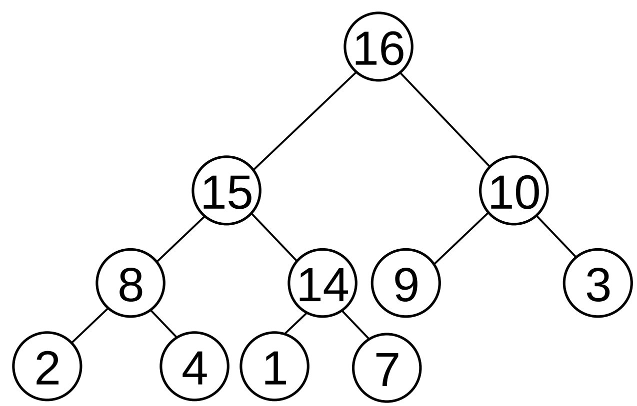


Increase the key to 15
Call HEAP-INCREASE-KEY on A[11] = 15



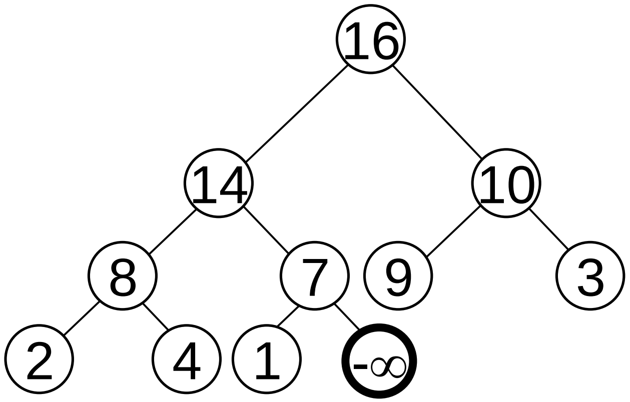


The restored heap containing the newly added element



Alg: MAX-HEAP-INSERT(A, key, n)

1. heap-size[A] ← n + 1
2. A[n + 1] ← -∞
3. HEAP-INCREASE-KEY(A, n + 1, key)



Running time: $O(\lg n)$

Summary

- We can perform the following operations on heaps:
  - MAX-HEAPIFY $O(\lg n)$
  - BUILD-MAX-HEAP $O(n)$
  - HEAP-SORT $O(n\lg n)$
  - MAX-HEAP-INSERT $O(\lg n)$
  - HEAP-EXTRACT-MAX $O(\lg n)$
  - HEAP-INCREASE-KEY $O(\lg n)$
  - HEAP-MAXIMUM $O(1)$

Average $O(\lg n)$